13. Why is “catch(Exception)” almost always a bad idea (and when it is not?)?

Brief summary: Using “catch(Exception)” should be avoided, because it catches every kind of exception. When we decide to catch an exception, we should know how to handle it, and it’s not feasible if the exception’s type is unknown. The acceptable use cases for catching any type of exceptions are:

- The global catch block that is catching all exceptions not handled elsewhere and shows them to the user.
- Any catch block in which we rethrow an exception without handling it.

Catching an object of System.Exception type, so the most general type of exceptions in C#, is almost always considered a bad idea. This is because when you catch an exception, you should handle it appropriately. But if you don’t know what the exception type is exactly, how can you know how to handle it?

Let’s consider the following code:

```
private static T? GetFirstOrDefault<T>(IEnumerable<T> items)
{
    try
    {
        return items.First();
    }
    catch (Exception)
    {
        Console.WriteLine("The collection is empty!");
        return default(T);
    }
}
```

This method works similarly to the parameterless FirstOrDefault method from LINQ - it returns the first element from the collection, but if the collection is empty, it returns the default value.

In the catch clause, we catch any exception and we print the information that the collection is empty. But other exceptions, not related to the emptiness of the

collection, can be thrown in the try clause, like OutOfMemoryException or StackOverflowException. For those two types, it is extremely hard to predict when they will happen (of course, in a healthy application they shouldn't happen at all, but we don't know how the rest of the application looks like). It may be the case that the catch clause will catch OutOfMemoryException, and it will handle it with the false message, saying that the collection is empty, even if it wasn't. The true problem - the lack of memory in the application - will be swept under the rug.

Because of that, we should always catch as specific exceptions as possible:

```
private static T? GetFirstOrDefault<T>(IEnumerable<T> items)
{
    try
    {
        return items.First();
    }
    catch (InvalidOperationException)
    {
        Console.WriteLine("The collection is empty!");
        return default(T);
    }
}
```

The First method throws InvalidOperationException, so handling it here is most appropriate.

For the same reasons, we should throw as specific exceptions as possible from our own code. Let's consider this method:

```
private static double Average(IEnumerable<int> numbers)
{
    if(numbers == null)
    {
        throw new Exception("The collection is null!");
    }
    double sum = 0;
    int count = 0;
    foreach (var number in numbers)
    {
        sum += number;
        count++;
    }
    if(count > 0)
    {
        return sum / count;
    }
    throw new Exception("The collection is empty!");
}
```

Throwing an Exception here is not a good idea. We should be more precise when choosing an exception type, so it fits the situation. When the first exception is thrown, the problem lies in the null collection passed to the Average method, so ArgumentNullException is a perfect fit. In the second case, when the collection is not null, but empty, InvalidOperationException or ArgumentException seem most appropriate:

```
private static double Average(IEnumerable<int> numbers)
{
    if(numbers == null)
    {
        throw new ArgumentNullException(
            "The numbers collection is null!");
    }
    double sum = 0;
    int count = 0;
    foreach (var number in numbers)
    {
        sum += number;
        count++;
    }
    if(count > 0)
    {
        return sum / count;
    }
    throw new InvalidOperationException(
        "The collection is empty!");
}
```

Exceptions give us priceless insight into what problems do we have in our application, and we should never mask them with some generic handling that will hide the detailed information they carry.

So is it ever appropriate to have the catch(Exception) clause?

Well, it is, in two specific scenarios.

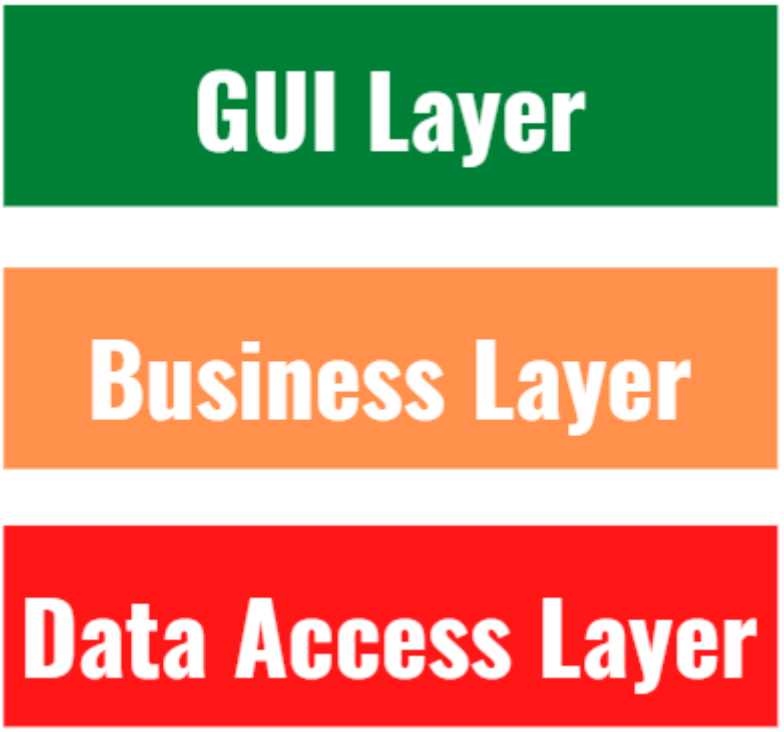
The first one is **the global catch block** in our application. This is the last resort of exception handling. If something totally unexpected happens and there is no way of continuing the application's work after that, we should simply show the

exception to the user and/or store it in some logs. After the user reads the error, the application will be stopped. Let's add a global catch block to a console application:

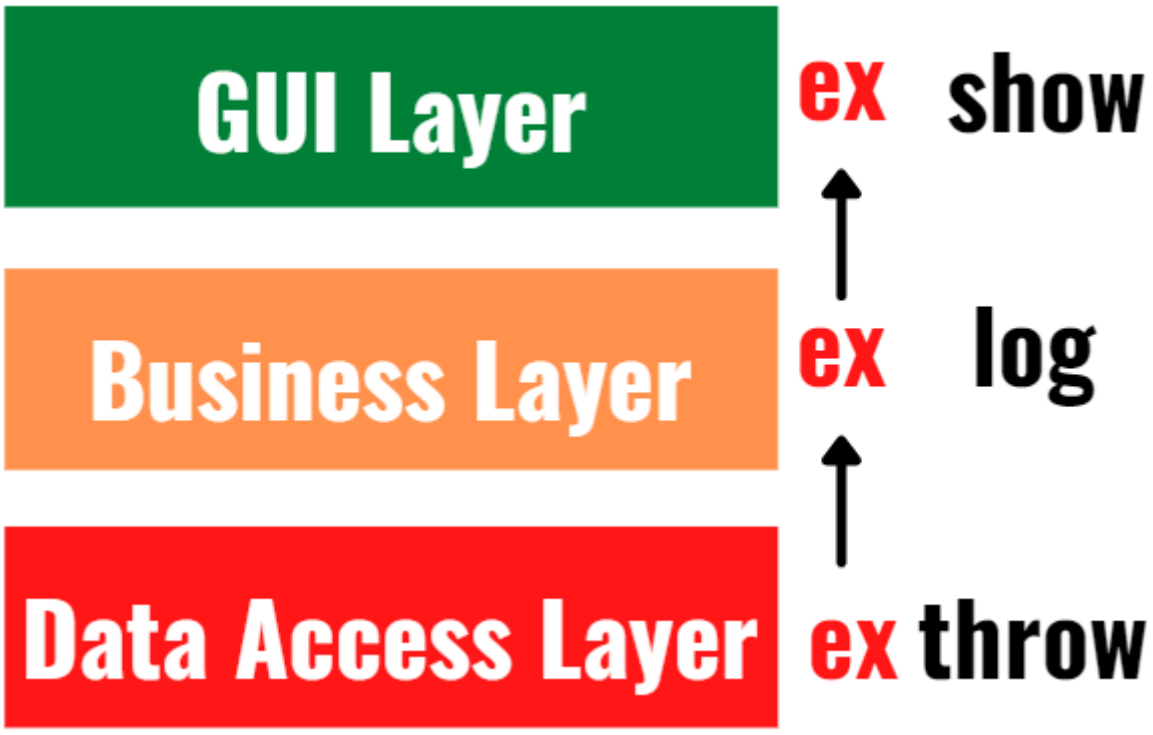
```
public static void Main(string[] args)
{
    try
    {
        var numbers = Enumerable.Empty<int>();
        var first = GetFirstOrDefault(numbers);

        //let's make the application crash on purpose:
        throw new Exception("Unknown exception");
    }
    catch (Exception ex)
    {
        //writing to some application logs would be appropriate here
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine($"An unexpected error happened and " +
            $"the application can't continue. The error message is" +
            $" '{ex.Message}', stack trace is : {ex.StackTrace}");
    }
    Console.ReadKey();
}
```

In applications with some proper GUI, such global catch block usually shows some error popup.

Another case when catching any type of exception could be OK is when we don't intend to handle it - we only rethrow it, possibly log it, or add some additional information to it. In such a tiny application as ours, it doesn't make much sense, but in big projects, it's often the case that applications are multi-layered, and each layer has its own way of reporting and organizing errors. This way, an exception thrown in the lower layer will be logged in each layer it crosses, but finally, it will be handled in some of the upper layers. Let's consider the following architecture:



If the exception is thrown at the Data Access Layer, it can be intercepted in the Business Layer, which logs it and then rethrows it. It is then handled in the GUI Layer by showing some popup with an error message to the user:



We will talk more about how to rethrow exceptions in the next lecture. In a simplified way, such code could look like this:

```
public class DataAccessLayer
{
    private readonly ILogger _logger;

    0 references
    public DataAccessLayer(ILogger logger)
    {
        _logger = logger;
    }

    1 reference
    public string GetRawData()
    {
        try
        {
            return " some raw data ";
        }
        catch (Exception ex)
        {
            _logger.LogError("Error in DataAccessLayer: " + ex);
            throw; //rethrowing the exception
        }
    }
}
```

This is the lowest-level layer. If an exception is thrown on data access, it is logged and rethrown. Later, it will be handled by the next layer - the Business Layer:

```
public class BusinessLayer
{
    private ILogger _logger;
    private DataAccessLayer _dataAccessLayer;

    0 references
    public BusinessLayer(
        DataAccessLayer dataAccessLayer,
        ILogger logger)
    {
        _dataAccessLayer = dataAccessLayer;
        _logger = logger;
    }

    1 reference
    public string GetProcessedData()
    {
        try
        {
            var rawData = _dataAccessLayer.GetRawData();
            return rawData.Trim();
        }
        catch(Exception ex)
        {
            _logger.LogError("Error in BusinessLayer: " + ex);
            throw; //rethrowing the exception
        }
    }
}
```

If Data Access Layer throws an exception here, or if the processing of the raw data does, the exception will be logged and rethrown. It will be truly handled (by showing some error to the user) in the upper-most layer - the GUI Layer:

```
public class GuiLayer
{
    private BusinessLayer _businessLayer;
    private ILogger _logger;

    0 references
    public GuiLayer(
        BusinessLayer businessLayer,
        ILogger logger)
    {
        _businessLayer = businessLayer;
        _logger = logger;
    }

    0 references
    public void ShowToUser()
    {
        try
        {
            var data = _businessLayer.GetProcessedData();
            Console.WriteLine($"Showing to user: {data}");
        }
        catch(Exception ex)
        {
            _logger.LogError("Error in GUI Layer: " + ex);
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"An unexpected error: '{ex.Message}'");
        }
    }
}
```

This way, the exception thrown at the lowest layer of the application will be logged at each layer, but it will only be handled at the GUI Layer. As you can see, at this point it is not rethrown, as this is the last place where it can be handled. Of course, if any of the lower layers could actually handle the exception and continue working without problem, it should not rethrow the exception, and it would never be shown in the error window.

Using “catch(Exception)” should be avoided, because it catches every kind of exception. When we decide to catch an exception, we should know how to handle it, and it’s not feasible if the exception’s type is unknown. We should be precise in both catching exceptions, as well as in throwing them. The acceptable use cases for catching any type of exceptions are:

- The global catch block that catches all exceptions not handled elsewhere, and shows them to the user.
- Any catch block in which we rethrow an exception without handling it.

Bonus questions:

- "**What are the acceptable cases of catching any type of exception?**"
  The acceptable use cases for catching any type of exceptions are:
  - *The global catch block that is catching all exceptions not handled elsewhere and shows them to the user.*
  - *Any catch block in which we rethrow an exception without handling it.*

- "**What is the global catch block?**"
  The global catch block is the catch block defined at the upper-most level of the application, that is supposed to catch any exceptions that hadn't been handled elsewhere. It usually logs the exception and shows some information to the user, before stopping the application.